Schichtenstrukturen

- Sprachen, Ebenen, virtuelle Maschinen
  - die „semantische Lücke“ erkennen und schließen bzw. überwinden
  - Übersetzung, Interpretation, partielle Interpretation
  - funktionale -, Modul-, Benutzthierarchie
- Hardware, Software, Mehrebenenmaschinen
  - digitale Logikebene, Mikroarchitekturebene, Befehlssatzebene
  - Maschinenprogrammebene ☞ Betriebssystemebene
  - Assembliersprachenebene, problemorientierte Sprachenebene
- Betriebssystem verstehen als virtuelle Maschine einer bestimmten Ebene

Semantische Lücke

semantic gap *The difference between the complex operations performed by high-level constructs and the simple ones provided by computer instruction sets. It was in an attempt to try to close this gap that computer architects designed increasingly complex instruction set computers.*[35]

- die Verschiedenheit, die zwischen *Quellsprache* und *Zielsprache* definiert ist
  - als Faustregel: $\left\{ \begin{array}{c} \text{höheres} \\ \text{niedrigeres} \end{array} \right\}$ **Abstraktionsniveau** $\left\{ \begin{array}{c} \text{Quellsprache} \\ \text{Zielsprache} \end{array} \right\}$
- die Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem

[35] `http://www.hyperdictionary.com/computing/semantic+gap`

Matrix-Matrix Multiplikation (1) Problemskizze

Multiplikation von zwei 2×2 Matrizen:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Daraus lässt sich allgemein für $C = A \times B$ ableiten: $C_{i,j} = \sum_k A_{ik} \cdot B_{kj}$

Matrix-Matrix Multiplikation (2) Implementierung in C

```
#define N 2
typedef int Matrix [N][N];

void multiply (const Matrix a, const Matrix b, Matrix c) {
    unsigned int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Matrix-Matrix Multiplikation (3) — Umsetzung für x86

```
_multiply:
    pushl %ebp
    movl  %esp, %ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    subl  $12, %esp
    movl  $0, -16(%ebp)
    movl  16(%ebp), %edi
L16:
    movl  $0, -20(%ebp)
    movl  -16(%ebp), %eax
    movl  -16(%ebp), %ebx
    sall  $3, %eax
    addl  %ebx, %ebx
    movl  %eax, -24(%ebp)
    .align 16
L15:
    movl  $0, (%edi,%ebx,4)
    movl  -20(%ebp), %edx
```

```
    xorl  %esi, %esi
    movl  12(%ebp), %eax
    leal  (%eax,%edx,4), %ecx
    movl  8(%ebp), %eax
    movl  -24(%ebp), %edx
    addl  %eax, %edx
L14:
    movl  (%ecx), %eax
    incl  %esi
    addl  $8, %ecx
    imull (%edx), %eax
    addl  $4, %edx
    addl  %eax, (%edi,%ebx,4)
    cmpl  $1, %esi
    jbe L14
    incl  -20(%ebp)
    incl  %ebx
    cmpl  $1, -20(%ebp)
    jbe L15
    incl  -16(%ebp)
```

```
    cmpl  $1, -16(%ebp)
    jbe L16
    addl  $12, %esp
    popl  %ebx
    popl  %esi
    popl  %edi
    popl  %ebp
    ret
```


Matrix-Matrix Multiplikation (4) — Verschiedenheit

Ebene der Problemskizze .... 1 Summenformel

- welches Problem behandelt wird, ist (nahezu) offensichtlich
- eine semantische Lücke ist eigentlich nicht vorhanden

C-Ebene .... 5 Komplexschritte

- welches Problem behandelt wird, ist (für Experten) noch erkennbar
- die semantische Lücke ist vergleichsweise klein

x86-Ebene .... $43{+}n$ Maschinenanweisungen

- welches Problem behandelt wird, ist (auch für Experten) nicht erkennbar
- die semantische Lücke ist vergleichsweise sehr groß

Matrix-Matrix Multiplikation (5) — x86-Hexkode

```
5589E557565383EC0CC745F0000000008B7D10C745EC000000008B45F08B5DF0
C1E00301DB8945E8908DB42600000000C7049F000000008B55EC31F68B450C8D
0C908B45088B55E801C28B014683C1080FAF0283C20401049F83FE0176ECFF45
EC43837DEC0176C8FF45F0837DF00176A283C40C5B5E5F5DC3
```

☞ die Diskrepanz zwischen der vom Menschen skizzierten Problemlösung und dem dazu korrespondierenden, von der Maschine „x86“ ausführbaren Programm, ist beträchtlich:

$$C_{i,j} = \sum_k A_{ik} \cdot B_{kj} \overset{?}{\Longleftrightarrow} \texttt{5589E5 ... 5F5DC3}$$

Komplex „Rechensystem“

[Problem]

Hochsprache

$\Uparrow$

semantische Lücke

$\Downarrow$

CPU-Steuerinformationen

[Ausführung]

- die Größe der semantischen Lücke variiert

  bei gleich bleibendem Problem
  - mit der Plattform (dem System)

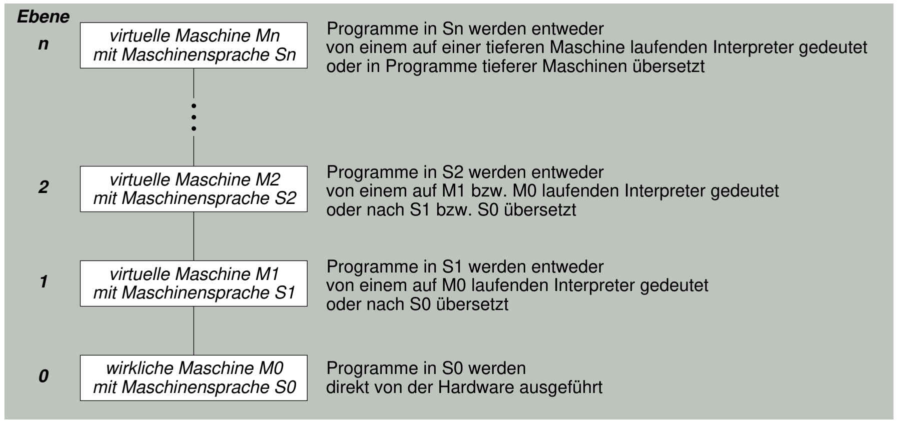
  bei gleich bleibender Plattform
  - mit dem Problem (der Anwendung)

- der Lückenschluss ist ganzheitlich zu sehen

- durch *schrittweise Abstraktion* die Kluft zwischen „oben“ und „unten“ schließen

- Poblemlösungen über **virtuelle Maschinen** auf die reale Maschine abbilden

Virtuelle Maschinen

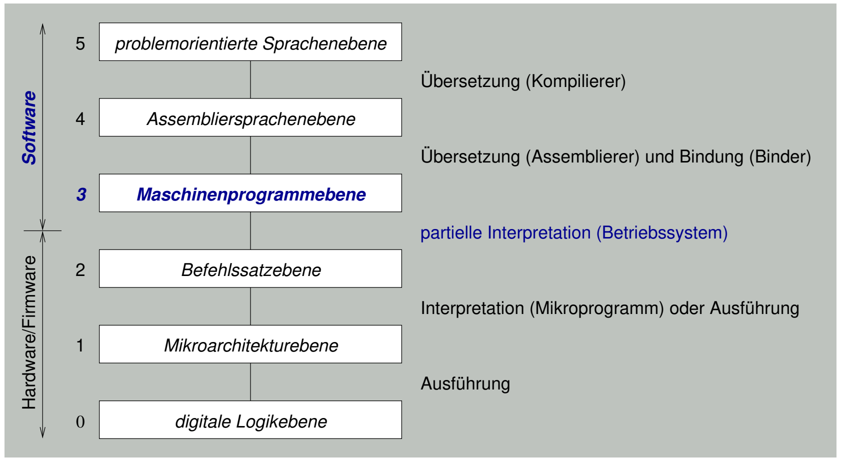


Abbildende Programme

Kompilierer (*compiler*) **Kom|pi|la|tor** *lat.* (Zusammenträger)
Ein typischerweise in Software realisierter *Prozessor*, der Programme einer bestimmten *Quellsprache* (z. B. C++) in semantisch äquivalente Programme einer bestimmten *Zielsprache* (z. B. C oder Assembler) transformiert.

Interpretierer (*interpreter*) **In|ter|pret** *lat.* (Ausleger, Erklärer, Deuter)
Ein in Hard-, Firm- oder Software realisierter *Prozessor*, der Programme einer bestimmten Quellsprache (z. B. Basic, Perl, C, `sh(1)`) „direkt“ ausführt. Bei *Vorübersetzung* durch einen Kompilierer werden die Programme zunächst in eine für die Interpretation günstigere Repräsentation (z. B. Pascal P-Code, Java Bytecode, x86-Befehle) transformiert.

Konventioneller Rechner



Softwaremaschinen

problemorientierte Sprachenebene „Höhere Programmiersprachen“ erlauben die abstrakte und Plattform-unabhängige Formulierung von Problemlösungen.

Assembliersprachenebene Pseudobefehle (Assemblierer/Binder), mnemonisch ausgelegte Maschinenbefehle (ISA) und symbolisch bezeichnete Operanden (Speicheradressen, Register) und Adressierungsarten bilden die Programme (*symbolischer Maschinenkode*).

Maschinenprogrammebene (**Betriebssystem**) Legt die Betriebsarten fest, verwaltet die Betriebsmittel des Rechners und steuert bzw. überwacht die Abwicklung von Programmen. *Systemaufrufe* und *Maschinenbefehle* (ISA) bilden die Elementaroperationen der Programme (*binärer Maschinenkode*).

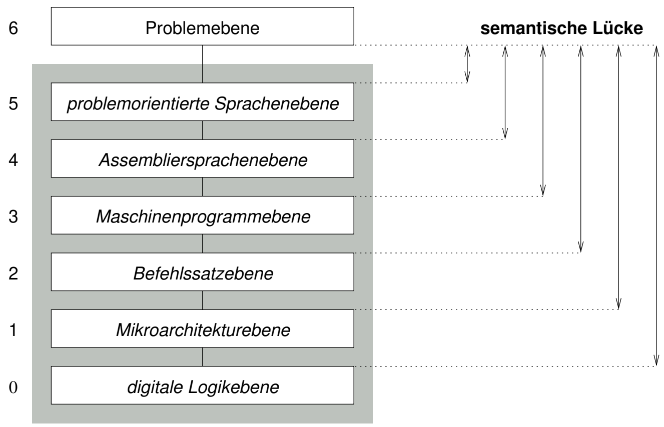
{Firm,Hard}waremaschinen

Befehlssatzebene (*instruction set architecture*, ISA) Implementiert das *Programmiermodell* der CPU (z. B. CISC, RISC, VLIW). Programme bestehen aus *Mikroanweisungen* oder Konstrukten einer *Hardwarebeschreibungssprache* (z. B. VHDL, SystemC).

Mikroarchitekturebene Beschreibt den Aufbau der Operations-/Steuerwerke, der Zwischenspeicher und die Befehlsverarbeitung. Programme bestehen aus Konstrukten einer *Hardwarebeschreibungssprache* (z. B. VHDL, SystemC).

digitale Logikebene Bildet auf Basis von Transistoren, Gattern, Schaltnetzen und Schaltwerken die wirkliche Hardware des Rechners. Programme bestehen aus Elementen der *Boolschen Algebra*.

Abstraktionsniveau vs. Semantische Lücke



(Software) Ebene $_5$ $\Longrightarrow$ Ebene $_4$ $\Longrightarrow$ Ebene $_3$ (Software)

Ebene $_5$ $\Longrightarrow$ Ebene $_4$ Kompilierung

- Ebene $_5$-Befehle „1:N" in *semantisch äquivalente* Ebene $_4$-Befehle übersetzen
  - ein Hochsprachenbefehl ist eine Folge von Assembliersprachenbefehlen
- im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

Ebene $_4$ $\Longrightarrow$ Ebene $_3$ Assemblierung und Binden

- Ebene $_4$-Befehle „1:1" in Ebene $_3$-Befehle übersetzen
  - ein Assembliersprachenbefehl ist ein Mnemonik
- jedes *Quellmodul* in ein korrespondierendes *Objektmodul* umwandeln
  - Objektmodule enthalten noch „unfertige" Maschinenprogramme
- Objektmodule mit *Bibliotheken* zum Maschinenprogramm zusammenbinden

Problemorientierte Sprachenebene

`myecho.c`

```
main () {
    char c;
    while (write(1, &c, read(0, &c, 1)) != -1);
}
```

Der Systemaufruf `read(2)` überträgt ein Zeichen von Standardeingabe (0) an die Speicheradresse `&c`, deren Inhalt anschließend mit dem Systemaufruf `write(2)` zur Standardausgabe (1) gesendet wird. Die Schleife terminiert durch Unterbrechung, unter UNIX z. B. nach Eingabe von `^C`.

Assembliersprachenebene (1)

`myecho.s` (generiert mit „`gcc -O6 -S myecho.c`“)

```
main:
    pushl %ebp
    movl  %esp, %ebp
    pushl %esi
    pushl %ebx
    subl  $16, %esp
    leal  -9(%ebp), %ebx
    andl  $-16, %esp
    movl  %ebx, %esi
    .align 16
.L2:
    movl  %esi, 4(%esp)
    movl  $1, %edx
    movl  %ebx, %esi
    movl  %edx, 8(%esp)
    movl  $0, (%esp)
    call  read
    movl  %eax, 8(%esp)
    movl  %ebx, 4(%esp)
    movl  $1, (%esp)
    call  write
    incl  %eax
    jne .L2
    leal  -8(%ebp), %esp
    popl  %ebx
    popl  %esi
    popl  %ebp
    ret
```


Assembliersprachenebene (2)

`libc.a` (Auszüge aus der C-Bibliothek des `gcc(1)`-Kompilierers)[36]

```
read:
    push %ebx
    movl 16(%esp), %edx
    movl 12(%esp), %ecx
    movl 8(%esp), %ebx
    mov  $3, %eax
    int  $0x80
    pop  %ebx
    cmp  $-4095, %eax
    jae  __syscall_error
    ret
```

```
write:
    push %ebx
    movl 16(%esp), %edx
    movl 12(%esp), %ecx
    movl 8(%esp), %ebx
    mov  $4, %eax
    int  $0x80
    pop  %ebx
    cmp  $-4095, %eax
    jae  __syscall_error
    ret
```

```
__syscall_error:
    neg %eax
    mov %eax, errno
    mov $-1, %eax
    ret

    .comm errno,16
```

[36] Genauer gesagt wurden diese Programmsequenzen mit dem `disassemble`-Kommando des `gdb(1)` generiert, nachdem das (Linux/x86) Maschinenprogramm mit „`gcc -O6 -static -o myecho myecho.c`“ erzeugt worden ist.

Betriebssystemebene (1)

Linux (`kernel-source-2.4.20/arch/i386/kernel/entry.S`)

```
system_call:
    pushl %eax
    cld
    pushl %es
    pushl %ds
    pushl %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    ...
    cmpl  $(NR_syscalls),%eax
    jae   badsys
    call  *sys_call_table(,%eax,4)
    movl  %eax,24(%esp)
ret_from_sys_call:
    ...
    popl  %ebx
    popl  %ecx
    popl  %edx
    popl  %esi
    popl  %edi
    popl  %ebp
    popl  %eax
    popl  %ds
    popl  %es
    addl  $4,%esp
    iret
badsys:
    movl  $-ENOSYS,24(%esp)
    jmp   ret_from_sys_call
```


Betriebssystemebene (2)

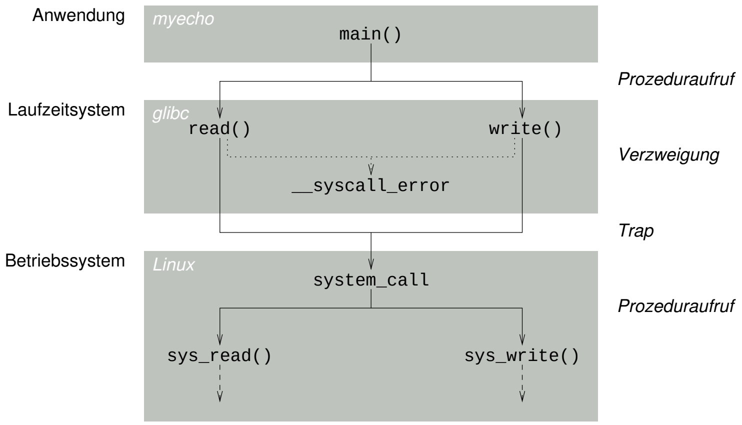
Linux (`kernel-source-2.4.20/fs/read_write.c`)

```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count)
{
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        ⋮
    }
    return ret;
}

asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
{
    ⋮
}
```

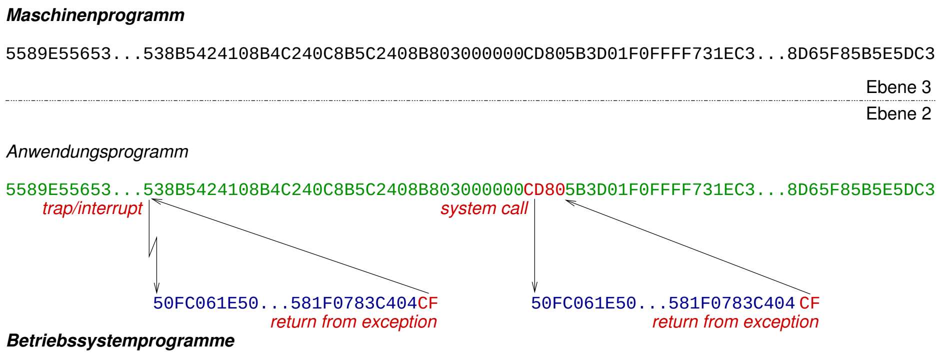
Softwaresystem „`myecho`“ — Aufrufhierarchie



(Software) Ebene $_3$ $\Longleftrightarrow$ Ebene $_2$ (Hardware)

- Maschinenprogramme setzen sich aus zwei Sorten von Befehlen zusammen:
  1. Aufrufe an das Betriebssystem
     - (a) explizit als Systemaufruf (*system call*) kodiert
     - (b) implizit als Programmunterbrechung (*trap*, *interrupt*) ausgelöst
  2. Anweisungen an die CPU
- ausführende Instanz im Rechner ist letztlich immer die CPU (Ebene $_{0-2}$), d. h.:
  - ☞ 1a, eine Art „Prozeduraufruf“ } schaltet um zum Betriebssystem
  - ☞ 1b, eine Ausnahmesituation }
- das Betriebssystem selbst ist ein „normales“ Programm von Ebene $_2$-Befehlen

Partielle Interpretation (1)



Partielle Interpretation (2)

- Ebene $_3$ ist eine *hybride Ebene* **Betriebssystemebene**
  - die meisten Befehle dieser Ebene sind Befehle der Ebene $_2$
  - zusätzliche Befehle implementieren z. B. Adressräume, Dateien, Prozesse
  - der Interpreter dieser Befehle ist das Betriebssystem
- die Aktivierung des Betriebssystems ist . . .
  - ☞ programmiert . . . . Systemaufruf (`CD80`)
  - ☞ nicht programmiert . . . . Ausnahmesituation
- die Deaktivierung des Betriebssystems ist immer programmiert (`CF`)

Systemaufruf (*system call*)

- je nach *Rechnerkonzept* sind die Systemaufrufe unterschiedlich ausgelegt:

  multiformer Betrieb Unterscheidung verschieden privilegierter Arbeitsmodi
  - Programme auf $\left\{\begin{array}{c}\text{Ebene}_3\\ \text{Ebene}_2\end{array}\right\}$ laufen im $\left\{\begin{array}{c}\text{nicht-}\\ \text{---}\end{array}\right\}$ privilegierten Modus
  - Systemaufrufe wechseln vom nicht-privilegierten zum privilegierten Modus
  - Rückkehr von Systemaufrufen reaktiviert den nicht-privilegierten Modus

  uniformer Betrieb für alle Programme einheitlicher Arbeitsmodus [37]
  - Systemaufrufe treten als konventionelle Prozeduraufrufe in Erscheinung

- die Ebene $_2$-Maschine (CPU) unterstützt beide Betriebsformen entsprechend:

  x86 ☞ [`int`/`iret` bzw. `call`/`ret`], **m68k** ☞ [`trap`/`rte` bzw. `jsr`/`rts`]

[37] Implementiert die CPU verschiedene Modi, geschieht der uniforme Betrieb im privilegiert(est)en Arbeitsmodus.

Unterbrechungsarten

- zwei Kategorien von Programmunterbrechungen werden unterschieden:
  1. die „Falle“ — der **Trap**
  2. die „Unterbrechung“ — der **Interrupt**

- die sie auslösenden Ausnahmesituationen unterscheiden sich hinsichtlich . . .
  - Quelle
  - Synchronität
  - Vorhersagbarkeit
  - Reproduzierbarbeit

- eine Behandlung ist zwingend und grundsätzlich prozessorabhängig[38]

[38] Egal ob abstrakter Prozessor (virtuelle Maschine) oder physikalischer Prozessor (reale Maschine).

Trap vs. *Interrupt*

Trap synchron, vorhersagbar, reproduzierbar . . . . *kein Interrupt*

- unbekannter Befehl, falsche Adressierungsart, fehlerhafte Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät (*bus fault*)

  ⋮
- Seitenfehler im Falle lokaler Ersetzungsstrategien

Interrupt asynchron, unvorhersagbar, nicht reproduzierbar . . . . *kein Trap*

- Signalisierung „externer“ Ereignisse
- Beendigung einer DMA- bzw. E/A-Operation

  ⋮
- Seitenfehler im Falle globaler Ersetzungsstrategien

Synchrone Programmunterbrechung

- der **Trap** — ist vorhersagbar und reproduzierbar

  Ein in die Falle gelaufenes („ge*trap*ptes“) Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und dem selben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle tappen, d. h. den selben Trap verursachen.

- ohne Behebung der Ausnahmebedingung ist eine Trap-Vermeidung unmöglich

Asynchrone Programmunterbrechung

- der **Interrupt** — ist unvorhersagbar und nicht reproduzierbar

  Auch wenn manche Geräte (z. B. Zeitgeber) Interrupts in gleichen zyklischen Zeitabständen (re-)produzieren können, ist die Stelle der Unterbrechung (d. h. der unterbrochene Maschinenbefehl) nicht vorhersagbar. Interrupts sind jedoch vorhersehbar in dem Sinne, dass (je nach Systemkonfiguration) mit ihrem Auftreten zu rechnen ist.

- die Behandlung der Ausnahmesituation muss <u>nebeneffektfrei</u> verlaufen

¿ *Trap* oder *Interrupt* ?

```
#include <stdlib.h>

float frandom () {
    return random()/random();
}
```

- eine (zufällige) Division durch 0 ist möglich
  - dies führt zur Programmunterbrechung
  - je nach CPU (bzw. ALU)
- ist diese Unterbrechung unvorhersagbar?
- es liegt ein (sich allerdings evtl. niemals auswirkender) *Programmierfehler* vor
  - die Stelle der (zufälligen) Programmunterbrechung ist vorhersagbar
  - ohne Fehlerbeseitung ist diese Unterbrechung somit reproduzierbar
  - die Unterbrechung ist synchron zur Programmausführung

☞ *Trap*

¿ *Trap* oder *Interrupt* ?

```
extern edata, end;

int main () {
    char* p = (char*)&edata;
    do *p++ = 0;
    while (p != (char*)&end);
}
```

- ein Seitenfehler (*page fault*) ist möglich
  - dies führt zur Programmunterbrechung
  - je nach Auslastung des Arbeitsspeichers
- er ist (un)vorhersagbar, (un)reproduzierbar
- die *Ersetzungsstrategie* des Betriebssystems bestimmt die Unterbrechungsart:

lokale Seitenersetzung	⇒	synchrone Programmunterbrechung
globale Seitenersetzung	⇒	asynchrone Programmunterbrechung

☞ *Trap* oder *Interrupt*

Ausnahmesituationen

- oftmals unerwünschte, aber auch nicht immer eintretende Ereignisse:
  - Signale von der Peripherie .... (z. B. E/A, Zeitgeber oder „Wachhund“)
  - Wechsel der Schutzdomäne .... (z. B. Systemaufruf)
  - Programmierfehler .... (z. B. ungültige Adresse)
  - unerfüllbare Speicheranforderung .... (z. B. bei Rekursion)
  - Warnsignale von der Hardware .... (z. B. Energiemangel)
  - Seitenfehler (*page fault*) .... [warum?]
- zu erwartende Ereignisse, die problemspezifisch zu behandeln sind

Ausnahme — *Exception*

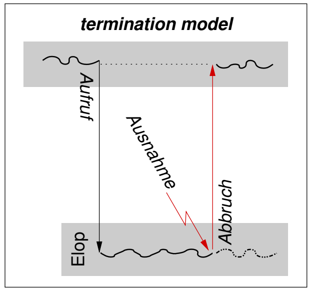
- aus Verursachersicht sind grundlegend zwei Konzepte zu unterscheiden [20]:

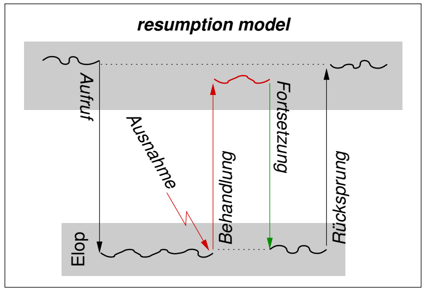
  resumption model Die erfolgreiche Behandlung der Ausnahmesituation führt zur Wiederaufnahme der Ausführung des unterbrochenen Programms. Ein Trap kann, ein Interrupt muss nach diesem Modell behandelt werden.

  termination model Konnte (oder sollte) die Ausnahmesituation nicht behandelt werden, wird ein schwerwiegender Fehler konstatiert, der zum Abbruch des unterbrochenen Programms führen muss. Ein Trap kann, ein Interrupt darf niemals nach diesem Modell behandelt werden.

- auslösen (*raising*) einer Ausnahme impliziert einen **Kontextwechsel**

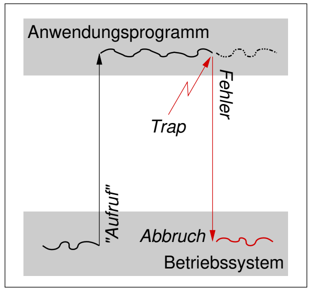
Ausnahmebehandlung — *Exception Handling*

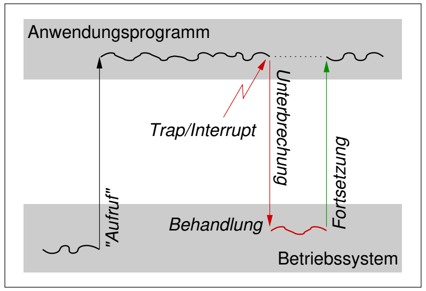




Elop — *El*ementar*op*eration eines (abstrakten) Prozessors/einer (virtuellen) Maschine: der Maschinenbefehl der CPU, eine Betriebssystemfunktion (aktiviert via *system call*), ein Unterprogramm (aktiviert via Prozeduraufruf).

Ausnahmebehandlung — *Traps/Interrupts*





Abrupter Zustandswechsel

- Programmunterbrechungen ziehen *nicht-lokale Sprünge* nach sich . . .

$$\text{vom} \left\{ \begin{array}{c} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\} \text{Programm zum} \left\{ \begin{array}{c} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\} \text{Programm}$$

- der **Prozessorstatus** des unterbrochenen Programms gilt dabei als Invariante
  - dies erfordert Maßnahmen zur Sicherung/Wiederherstellung des Kontextes
  - die Mechanismen liefert das behandelnde Programm bzw. eine tiefere Ebene
- ein „tieferer“ realer und/oder abstrakter Prozessor hält den Status konsistent

Prozessorstatus — Kontext

- die CPU (Ebene $_2$) sichert einen Zustand <u>minimaler Größe</u>
  - typischerweise Statusregister (SR) und Befehlszeiger (*program counter*, PC)
  - möglicherweise aber auch den kompletten Registersatz des Prozessors
  - je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt
- das Betriebssystem (Ebene $_3$) sichert den restlichen Zustand
  - d. h., alle $\left\{ \begin{array}{c} \text{dann noch ungesicherten} \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\}$ CPU-Register
- die Sicherungsmaßnahmen sind höchst abhängig vom jeweiligen Prozessor

Prozessorstatus Sicherung und Wiederherstellung

```
tip:
  moveml d0-d7/a0-a6,a7@-
  jsr    handler
  moveml a7@+,d0-d7/a0-a6
  rte
```

```
void __attribute__ ((interrupt)) tip () {
    handler();
}
```

„***tip***“ (*trap/interrupt plug*) mit z. B. zwei Lösungsoptionen:

o. links: Maschinenbefehle programmieren, die *alle noch nicht gesicherten* (m68k) CPU-Register konsistent halten

o. rechts: Maschinenbefehle generieren, die *alle im weiteren Verlauf verwendeten* CPU-Register konsistent halten

Beide Varianten nutzen dieselbe Behandlungsprozedur (rechts).

```
inline void handler() {
    ...
}
```


„Wettkampfrisiko“ — Überlappung, Nebenläufigkeit

race hazard (auch *race condition*)[39]

- fehlerhafte Stelle in einem System, an der eine Berechnung eine unerwartet kritische Abhängigkeit vom relativen Zeitverlauf von Ereignissen zeigt
- Softwaresysteme sind mit dem Problem konfrontiert, sobald die Ausführung von Programmen überlappend, nebenläufig oder parallel möglich ist
  - während ein Prozess den Platzhalter einer Variablen ausliest (beschreibt), beschreibt (liest) ein anderer Prozess denselben (aus)
  - typischer Fall: die Behandlung asynchroner Programmunterbrechungen
- Zugriffe auf gemeinsame Variablen hinterlassen ggf. undefinierte Zustände

[39] Dem Begriff liegt die Vorstellung zu Grunde, dass sich zwei (oder mehr) Signale in einem Wettlauf zueinander befinden, um als erstes die Ausgabe (ein Berechnungsergebnis) zu veranlassen.

Nebenläufiges Zählen (1)

```
unsigned int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    wheel++;
}

int main () {
    for (;;)
        printf("%10u", wheel++);
}
```

```
main: ...
.L2:
    movl  wheel, %edx
    incl  %edx
    movl  %edx, wheel
    ...
    jmp   .L2

tip:
    pushl %eax
    movl  wheel, %eax
    incl  %eax
    movl  %eax, wheel
    pushl %eax
    iret
```

☞ Welche `wheel`-Werte werden ausgegeben?

Nebenläufiges Zählen (2)

main()		tip()		wheel
x86-Befehl	`%edx`	*x86-Befehl*	`%eax`	
`movl wheel,%edx`	42			42
		`movl wheel,%eax`	42	42
		`incl %eax`	43	42
		`movl %eax,wheel`	43	43
`incl %edx`	43			43
`movl %edx,wheel`	43			43

☞ zweimal wurden die Zählanweisungen durchlaufen, aber nur einmal wurde `wheel` inkrementiert

☞ Problem (im vorliegenden Fall): Zählen ist *keine atomare Operation*

Atomares Zählen (1)

- Zählen als *Elementaroperation* (**Elop**) eines abstrakten Prozessors auffassen
  - eine Elop ist per Definition eine *atomare Operation*
  - die *Unteilbarkeit* sichert der die Elop implementierende Prozessor
- als Lösung (für Monoprozessorsysteme) bieten sich folgende Ansätze an:

  Komplexbefehle der CPU verwenden (nur bei CISC)

```
movl wheel, %edx
incl %edx
movl %edx, wheel
```

$\Longrightarrow$

```
incl wheel
```

  temporäres Abschalten von Interrupts (*disable/enable interrupts*)

Atomares Zählen (2)

```
inline int incr (int* ip) {
    asm ("incl %0" : : "g" (*ip));
    return *ip;
}
```

```
unsigned int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    incr(&wheel);
}

int main () {
    for (;;) printf("%10u", incr(&wheel));
}
```

```
inline int incr (int* ip) {
    asm ("pushf");
    asm ("cli");
    *ip += 1;
    asm ("popf");
    return *ip;
}
```

o. links Komplexbefehl der CPU

o. rechts temporäres Abschalten von Interrupts: `cli` schaltet ab, `popf` stellt alten Status (`pushf`) her

Für x86 und mit *inline assembler* (`gcc`).

Selbstvirtualisierung (1)

- die CPU „*trappt*" privilegierte Befehle im nicht-privilegierten Arbeitsmodus
  - beim x86 z. B. Befehle
    - zur Ein-/Ausgabe (`in`, `out`)
    - zur Synchronisation (`cli`, `sti`)
    - zum Moduswechsel (`iret`)
    - ⋮
- das Betriebssystem wird durch eine Programmunterbrechung aktiviert
  - es emuliert den privilegierten Befehl „anwendungsgewahr"
  - kontrollierte Aktionen schaffen die Illusion vom direkten Hardwarezugriff
- nicht jede CPU unterstützt das Betriebssystem zur Emulation

Selbstvirtualisierung (2)

- die CPU/MMU „*trappt*“ unerlaubte Programmzugriffe auf den Arbeitsspeicher
  - jedem Programm ist ein eigener *logischer Adressraum* zugeordnet
  - das Betriebssystem bildet diesen ab auf den *phyiskalischen Adressraum*
  - die MMU garantiert die Integrität der Adressräume der Programme
- das Betriebssystem wird durch eine Programmunterbrechung aktiviert
  - es untersucht/behandelt die *Adressraumverletzung*
  - der Zugriff wird zurückgewiesen oder „anwendungsgewahr“ emuliert [40]
- nicht jede CPU ist mit Adressumsetzungshardware (MMU) bestückt

[40] Beispielsweise kann damit die Illusion von speicherabgebildeter Ein-/Ausgabe (*memory mapped I/O*) auch auf einer Hardware, die dieses Konzept nicht implementiert, aufrecht gehalten werden.

Emulation

- die **Nachahmung der Eigenschaften eines ggf. anderen Rechnersystems**
  - bei Selbstvirtualisierung emuliert sich ein Rechnersystem selbst
  - ein *Emulator* interpretiert die von der CPU nicht ausführbaren Befehle
- manche Rechnerarchitekturen definieren zusätzliche „*virtuelle Befehlssätze*“
  - d. h. Befehle, die optional in Hardware (Koprozessoren) implementiert sind
  - typischer Fall ist eine Fließkommaeinheit (*floating-point unit*, FPU) [41]
  - fehlt die Einheit, erfolgt eine Emulation ihrer Befehle in Software
- Ebene $_2$-Befehle sind Bestandteil der ISA, ihre Implementierungen nur bedingt

[41] Ebenso z. B. Graphikbeschleuniger, Adressumsetzungs- und Kommunikationshardware.

Hierarchien

- der Begriff „Hierarchie“ ist (nicht nur in der Informatik) mehrdeutig belegt:
  - einerseits wird die (physische) **Aufrufbeziehung** dargestellt
    - ☞ **funktionale Hierarchie**
    - ☞ **Modulhierarchie**
  - andererseits wird die (logische) **Abhängigkeitsbeziehung** angegeben
    - ☞ **Benutzthierarchie** [21]
- diese Arten von Hierarchie führen zu unterschiedlichen Repräsentationen
  - welche Art ausgewählt wird, hängt davon ab, was ausgedrückt werden soll

Funktionale Hierarchie

- die Anordnung beschreibt den *funktionalen Zusammenhang* von Programmen
  - definiert wird die *logische Beziehung*, nicht zwingend die physikalische
  - das Speicherabbild (*memory footprint*) muss keine Hierarchie mehr zeigen
- der Entwurf abstrahiert von der konkreten Repräsentation der Funktionen
  - eine Ausprägung kann erfolgen als Makro, Unterprogramm oder Prozess
    - ∗ ebenso beliebige Kombinationen davon
  - die Implementierungsform wird zur Konfigurierungs- bzw. Generierungsfrage
- die Implementierung enthält keine Funktion, die der Entwurf nicht zeigt

Modulhierarchie

***Information modules** are comprised of some data structures (possibly) and a set of functions which share knowledge of a particular design decision (reflected, for example, in the details of the data structures).* [22]

- beschrieben wird der *logische Zusammenhang* von Programm-Modulen
  - das Speicherabbild (*memory footprint*) muss keine Hierarchie mehr zeigen
  - gleiches gilt auch in Bezug auf die einzelnen Funktionen der Module
- die Programm-Module können sich über mehrere Schichten erstrecken
  - indem die Funktionen eines Moduls verschiedenen Schichten zugeordnet sind
  - eine Schicht definiert sich über Funktionen, nicht Module

Benutzthierarchie

- die Anordnung dokumentiert die *formale Abhängigkeit* von Programmen
  - sie beschreibt übergreifende Korrektheitsannahmen bzw. -anforderungen
  - in Beziehung werden gebracht Prozeduren, Module oder Programm(teil)e
- „benutzt" bedeutet „abhängig zu sein von einer korrekten Implementierung"
  - Programm A benutzt B, wenn. . .
    - ☞ zur Erfüllung von A's Aufgabe B's korrekte Ausführung zwingend ist
    - ☞ die Korrektheit von A von der Korrektheit von B abhängt
- dass A B benutzt, ist A's Spezifikation *und* Implementierung zu entnehmen

Aufrufbeziehung vs. Benutztbeziehung

- ein (Prozedur/Modul) Aufruf drückt nicht immer eine Benutztbeziehung aus
  - entsprechend A's Implementierung erfolgt der Aufruf von B nur bedingt:

```
int A (int x) { return (x > 42) ? 42 : (4711 / x); }
```

  - A könnte korrekt ablaufen, obwohl B's Implementierung inkorrekt ist
- eine Benutztbeziehung kann auch ohne explizite Aufrufe bestehen
  - A benutzt B implizit, wenn B Programmunterbrechungen behandelt
    - ☞ B (ggf. auch, dass B eintritt)[42] muss nebeneffektfrei für A sein
  - A könnte sonst scheitern, obwohl A's Impl. keinen Aufruf von B enthält

[42] Dass B eintritt, könnte zu Terminverletzungen bei A's Ausführung führen, was bei Echtzeitsystemen kritisch ist.

„Belegtes Butterbrot"

- eine Benutzthierarchie ist ein **azyklischer Graph** von Programm(teil)en:

  Ebene E_0 enthält Programme, die keine anderen Programme benutzen

  Ebene $E_{i,i>0}$ enthält Programme, die wenigstens ein Programm der Ebene E_{i-1}, aber kein Programm oberhalb Ebene E_{i-1} benutzen
- voneinander profitierende Programme führen zum Zyklus, der zu vermeiden ist
  - Annahme, A und B stünden in gegenseitiger Abhängigkeit:
    - ∗ zur Konfliktauflösung ist zu versuchen, etwa B in B_1 und B_2 so aufzuteilen, dass B_1 A und A B_2 benutzt
  - A wird „Belag" eines „belegten Brotes" mit den „Scheiben" B_1 und B_2
- das Aufspalten und Einschieben ist auch als „*sandwiching*" [21] bekannt

Fallstudie: Verdrängende Prozesseinplanung (1)

Angenommen, die vier wie folgt spezifizierten Funktionen sind gegeben:

alarm wird durch eine asynchrone Programmunterbrechung aktiviert und ruft `audit()` auf, um die Ausführung von Prozessen neu einzuplanen.

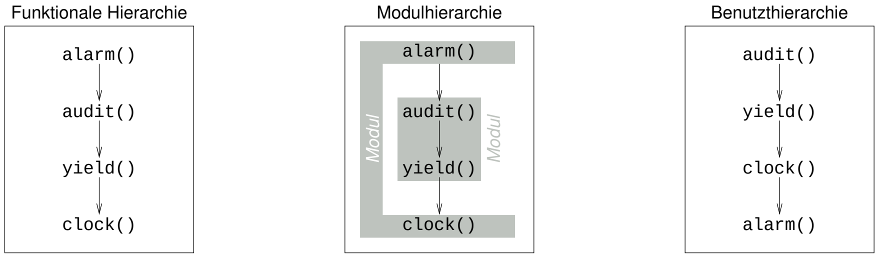
audit prüft, ob dem laufenden Prozess weiterhin die CPU zugeteilt bleibt und löst durch den bedingten Aufruf von `yield()` einen Prozesswechsel aus.

yield lässt sich zur Buchführung von Prozesslaufzeiten durch einen Aufruf von `clock()` einen Zeitwert liefern und schaltet zu einem anderen Prozess um.

clock liest die Hardwareuhr ab, berechnet die aktuelle Uhrzeit und liefert einen Zeitwert (z. B. in Nanosekundeneinheiten) zurück.

Wie lässt sich die Beziehung dieser Funktionen untereinander ausdrücken?

Fallstudie: Verdrängende Prozesseinplanung (2)



☞ `alarm()` und `clock()` teilen sich Wissen über die Uhr, `audit()` und `yield()` über die Implementierung von Prozessen

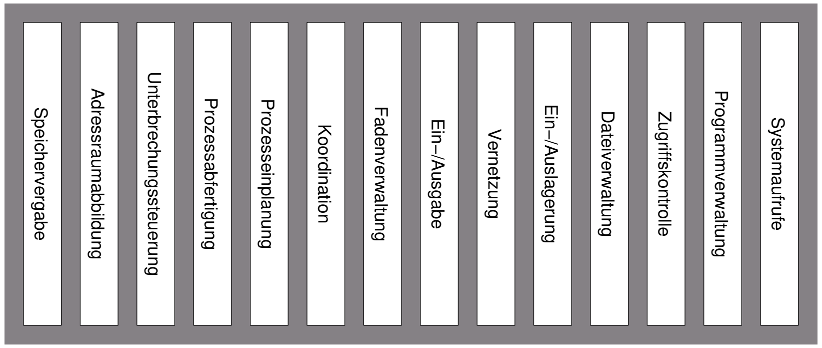
☞ `alarm()` kann jede Funktion unterbrechen und wird daher von allen benutzt

Betriebssystemmaschine

- Betriebssysteme (sollen) helfen, die semantische Lücke weiter zu verringern
  - sie befinden sich zwischen zwei Stühlen (Ebene $_2$ und $\geq$ Ebene $_4$)
  - eine besonders „exponierte Lage“, konfrontiert mit viel Konfliktstoff[43]
- ihre **logische Struktur** ist zuweilen recht komplex und auch sehr vielschichtig
  - sie orientiert sich an der (historisch gewachsenen) Gesamtfunktionalität
  - differenzierte Sichten entstehen mit unterschiedlichen *Betriebsarten*
- welche Schichten in welcher Weise vorliegen, ist vom realen Problem abhängig

[43] Betriebssysteme müssen „nach oben“ den verschiedensten Applikationsanforderungen und „nach unten“ den jeweiligen Hardwaregegebenheiten Rechnung tragen. Nicht immer sind beide Seiten in Einklang zubringen.

Verfeinerung von Ebene $_3$ — Benutzthierarchie



Logische/Physische Struktur

- Schichten sind logische und nicht physische (d. h., wirkliche) Strukturelemente
  - ihre Funktionen sind realisiert als Makros, Prozeduren, Module oder Prozesse

    Makro zur Übersetzungszeit an der Aufrufstelle expandierte Befehlsfolgen
    Prozedur an mehreren Stellen aufrufbares Unterprogamm als Unikat
    Modul Prozedur(en) mit gekapseltem (gemeinsamen) Datenbestand
    Prozess ein autonomer Kontrollfluss, ggf. mit eigenem Adressraum
  - entsprechend sind die Funktionsaufrufe technisch unterschiedlich ausgelegt
- im funktionalen Sinn ist jede Repräsentation einer Schicht gleich gut [23]

Betriebssystemarchitektur

Die „Erscheinungsform" der Betriebssystemmaschine wird mitbestimmt durch das Abkapselungsinstrument zur technischen Repräsentation der Systemfunktionen:

prozedurorientiert ☞ Makros und/oder Prozeduren; Systemaufruf bedeutet Makroexpansion bzw. Prozeduraufruf.

modulorientiert ☞ Module; Systemaufruf bedeutet Schutzdomänenwechsel (gestützt durch Sprachen und/oder Hardware [22]).

prozessorientiert ☞ Prozesse; Systemaufruf bedeutet Prozedur-Fernaufruf (*remote procedure call*, RPC [9, 24]) bzw. Prozesswechsel.

Mischformen sind gebräuchlich. Keine Form ist der anderen zwingend überlegen.

Modularisierung und Hierarchie

> *It is the system design which is hierarchical, not its implementation.*
>
> Habermann, 1976

Zusammenfassung

- zwischen Problemstellung und Rechnersystem klafft eine *semantische Lücke*
  - der *semantische Abstand* variiert mit dem Problem und der Hardware
  - auch das *Wissensniveau* der jeweiligen „Experten" ist ein Einflussfaktor
- die Hardware/Software-Hierarchie hilft, die semantische Lücke zu schließen
  - ein Rechner setzt sich demnach aus fünf grundsätzlichen Ebenen zusammen
  - eine Ebene ist repräsentiert als *virtuelle Maschine* bzw. *reale Maschine*
- die *Betriebssystemmaschine* ist eine besondere Art einer virtuellen Maschine
  - diese Ebene$_3$ bewerkstelligt die *partielle Interpretation* der Systemaufrufe
  - ihre Architektur variiert mit der jeweils zu unterstützenden Problemdomäne